Security Audit
Horizon Games
NiftySwap Exchange

Monday, 10-Feb-2020

Agustín Aguilar

Introduction

The Horizon Games team requested a security audit of their NiftySwap exchange project; the audited repositories follow:

https://github.com/arcadeum/niftyswap (commit 47f413e18226b1c0ac22cf488a1a0e6939b0bd81)

The contracts that constitute the project follow.

- NiftyswapExchange.sol: Uniswap-like dex implementation supports ERC1155 and uses callbacks to trigger trades; multiple IDs of the same ERC1155 can be traded simultaneously.
- NiftyswapFactory.sol: NiftyswapExchange factory contract that keeps track of created exchanges and avoids the creation of duplicated pools.

Issues

Critical severity

C1 - Pool is drainable if `base` matches `token`

The `NiftySwapExchange` contract allows anyone to create pools with any combination of `baseTokenID` and `tokenID`, opening the possibility for a pool of `base/base` to exist.

When the exchange rate for a pool is calculated, the contract assumes that all of its token balance corresponds to the pool, but this is not the case if the pair is `base/base`; in such a case, the balance of the contract includes the combined provided liquidity for all the other pairs.

This miscalculation of `_getTokenReserves` opens the possibility of executing two different attacks:

a) An attacker creates a pool of `base/base` and provides liquidity to it, then performs trades against such pool, consuming the base tokens of the other pools.
b) An attacker creates a pool of `base/base` tokens and provides liquidity to it, then uses the method `removeLiquidity()` to withdraw not only the provided base but also the liquidity of the other pools.

An example of how to replicate attack B follows:

- Create exchange with the same ERC1155 contract address for base and token, ID 0 as base
- User deposits 1000000000 ID 1 TOKEN and 1000000000 ID 0 BASE of liquidity, 0/1 pair
- Attacker obtains 1000000001 ID 0 tokens
- Attacker deposits 1 ID 0 TOKEN and 1000000000 ID 0 BASE of liquidity, 0/0 pair
- Total balance of pool ID 0 is 2000000000
- Attacker removes 666599999 liquidity tokens of the ID 0 pool
- Attacker balance of ID 0 becomes 1999799997, taking 999799996 extra ID 0 tokens

Low severity

L1 - Front-running during an exchange creation

The `NiftySwapFactory` contract limits the creation of exchanges to only one exchange for each token; however, a fixed `_baseTokenAddr` and `_baseTokenID` for each created exchange is not enforced.

Because the calling of `createExchange` is not permissioned, an attacker could create exchanges with random or broken `baseTokenAddr` or `baseTokenID` for all prominent ERC1155 tokens, blocking the genuine creation of those exchanges.

L2 - An error on `base` or `token` could leave the whole pool frozen

The `NiftySwapExchange` contract requires liquidity withdrawing to be of both in `base` and `tokens` at the same time.

In a scenario where any of the two ERC1155 token contracts encounters an error (and stops being transferable), the whole pool is affected, and the liquidity funds on the other token cannot be withdrawn.

A similar issue had happened to Uniswap exchange before, when Binance token didn't correctly implement `transferFrom`[1], and when the Synthetix migrated their token contract[2].

L3 - Minimum base provided can be circumvent

The `NiftySwapExchange` contract specifies a minimum of `1000000000` base tokens to be added during `addLiquidity`, (when the pool previously has no liquidity); this limit is in place to avoid the subsequent deposit having a significant rounding error.

An attacker could circumvent this limitation by adding liquidity above the limit and then withdrawing part of that liquidity, leaving the pool with less than `1000000000` base tokens of liquidity.

An example of how to replicate the issue follows:

- Attacker creates a pool with 1000000000 base 1000000000 token of liquidity
- Attacker removes 999999999 liquidity tokens
- The pool is left with only 1 base and 1 token of liquidity

[1] Uniswap Binance locked ETH (https://twitter.com/UniswapExchange/status/1072286773554876416)
[2] Synthetix token migration (https://blog.synthetix.io/guide-on-migrating-seth-liquidity/)

L4 - `addLiquidity` can round in favor of new liquidity provider

The `NiftySwapExchange` contract rounds up the requested `base` tokens during `addLiquidity` in order to favor the previously existing liquidity providers over the user adding liquidity; the rounded-up value is later used to determine the number of liquidity tokens that have to be minted in proportion to the existing pool.

Given that the proportion of `base` tokens added may be above the proportion of `tokens` added, in specific scenarios, the user adding liquidity may profit by adding a specific amount of liquidity.

An example follows:

- User deposits liquidity 1000 base and 100000 token
- Attacker adds liquidity, 1000 token, the proportional base is 10, is rounded to 11
- 11 Liquidity tokens are minted to the attacker
- Attacker withdraws 11 liquidity, in exchange for 11 base and 1098 token
- The user loses 98 token

Proposed solution:

a) Use the proportional `base` without rounding during the calculation of minted liquidity tokens.

Notes

N1 - Token trades aren't directly composable

When a trade is made, the bought tokens are transferred to the recipient's address with an empty `data` field, limiting the possibility of natively chaining multiple trades on a single transaction.

This issue could be partially fixed by enabling the users to customize the `_data` field when transferring the newly acquired tokens, which could encode an additional trade to be performed on another NiftySwap exchange.

It has to be noted that such a solution only enables trades between different exchange contracts, but trades with different IDs on the same exchange contract would still not be possible.

N2 - Rounding up can be improved

The `NiftyswapExchange` contracts tend to round transactions in a way that's favorable to the existing liquidity providers; this avoids situations where they pay for inefficiencies during trading or liquidity additions.

This rounding is performed by increasing the result of a division by `1`. This rule is also applied to divisions with a remainder of zero that don't require any rounding.

Consider replacing it with a more precise `divRound` method:

```
a / b + a % b == 0 ? 0 : 1
```

N3 - `functionSignature` retrieval can be simplified

The `onERC1155BatchReceived` method on the exchange uses assembly code to retrieve the first 4 bytes of the `_data` field; this code can be simplified by using `abi.decode(_data, (bytes4))`. Although it costs a little more gas, it's simpler to review and audit.

N4 - `self-deposit` isn't enforced

The exchange defines a function signature for the self-deposit of ERC1155. It's used during an `addLiquidty` operation to accept the pulled base tokens.

However, the exchange doesn't validate that the operator of the self-deposit transfer is, in fact, itself. It also doesn't validate that the self-deposited token is the base token.

This allows the deposit of any ERC1155 token on the exchange as long as the data contains the self-deposit signature.

N5 - Definition, and assignment of arrays can be optimized

The exchange contract always initializes the `tokensLiquidity` before assigning the value after calling `getTokensLiquidity`.

The two operations could be combined to reduce memory expansion and gas usage.

N6 - Functions mutability can be restricted to pure

The functions `getBuyPrice` and `getSellPrice` don't access the chain state, and thus their scope can be re-defined as `pure`.

N7 - `refundBase` isn't sent to the seller

During a base to token trade, the amount to be traded is specified by how many tokens are intended to be bought; because of possible exchange rate fluctuations, the seller is expected to send additional `base` tokens, which are refunded in case of not being used.

This refund is always sent to the `recipient` of the trade, which may not be the address originating the trade. The user originating the trade may be expecting the recipient to only receive the bought tokens, not any exceeding base tokens, as described by the `_baseToToken` documentation.

Proposed solutions:

a) Document the behavior, that the `recipient` also receives any exceeding `base` tokens.
b) Send any exceeding `base` tokens to the originator of the trade.

N8 - Minted liquidity tokens can round to zero

The `addLiquidity` method mints liquidity tokens proportionally to the provided `base` tokens. In specific scenarios, zero liquidity tokens may be minted to the liquidity provider.

It's reasonable to assume that a liquidity provider is expecting liquidity tokens to be minted in exchange for the provided liquidity; thus, the transaction should revert.

Final thoughts

The contracts composing the audited projects are well written and efficient.

The C1 vulnerability must be addressed before deploying the project in a production environment.

Curation of the displayed ERC-1155 tokens on the front-end is recommended, the project assumes that the traded tokens are a near perfect implementation of the ERC-1155 standard, and any miss-implementation or deviation could pose severe issues for the liquidity providers.

- February 2020 - Agustín Aguilar